

The Little Register Allocator

Yin Wang

The name of the game

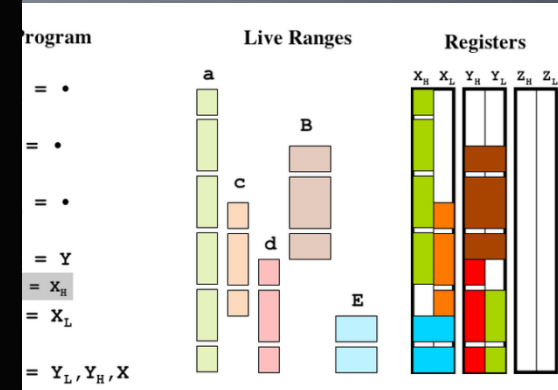
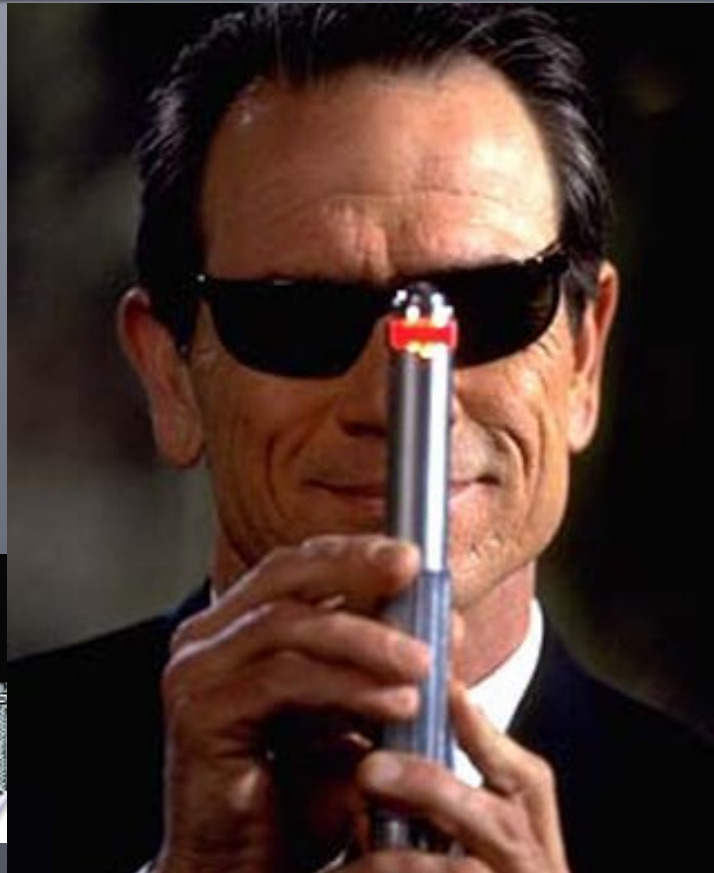
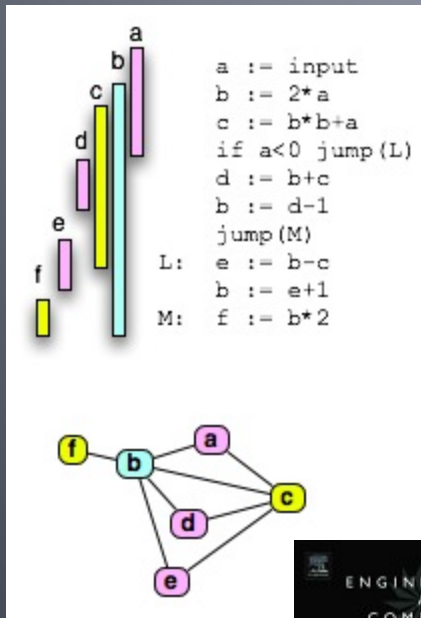


“See that bird? It's a Spencer's warbler. Well, in Italian, it's a Chutto Lapittida. In Portuguese, it's a Bom da Peida. In Chinese it's a Chung-Long-tah, and in Japanese it's a Katano Tekeda. You can know the name of the bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird. You'll only know about humans in different places, and what they call the bird. So let's look at the birds and see what it's doing -- that's what counts!”

-- Richard Feynman, *The Making of a Scientist*

The game of the name

Let's look at the register allocator and see what it's doing ...



Names, Registers, Stack

- Why do we need names in programs?
 - Because we want to refer to values.
-
- Why do processors have registers?
 - Because its circuits need to refer to ... values.
-
- So, are registers analogous to names?
 - Yes. They are of the same essence, holders of values.

- Can we use registers to stand for names?

- Yes. Because the values of names can also live in registers.

- Can I say that names can live in registers?

- You may say so, although it is their values that are living.

- Can ALL names live in registers?

- Not necessarily. Processors have very few registers, because they are expensive.

- Where do the rest of the names live?

- The stack.

- What is the stack?

- A large and cheap off-processor space where names can live.

- Can processors do arithmetic on the stack?

- No. They have to load the operands into registers first.

The Life of Names

A name lives either in an expensive register or a cheap stack slot.

Register Allocation

- What is register allocation?
 - The process of deciding **where** and **when** each name lives.
-
- Does it take a long time to move value between register and stack?
 - Yes. It is a major cost of time.
-
- What should be the goal of register allocation then?
 - To minimize register-stack traffic.

Model

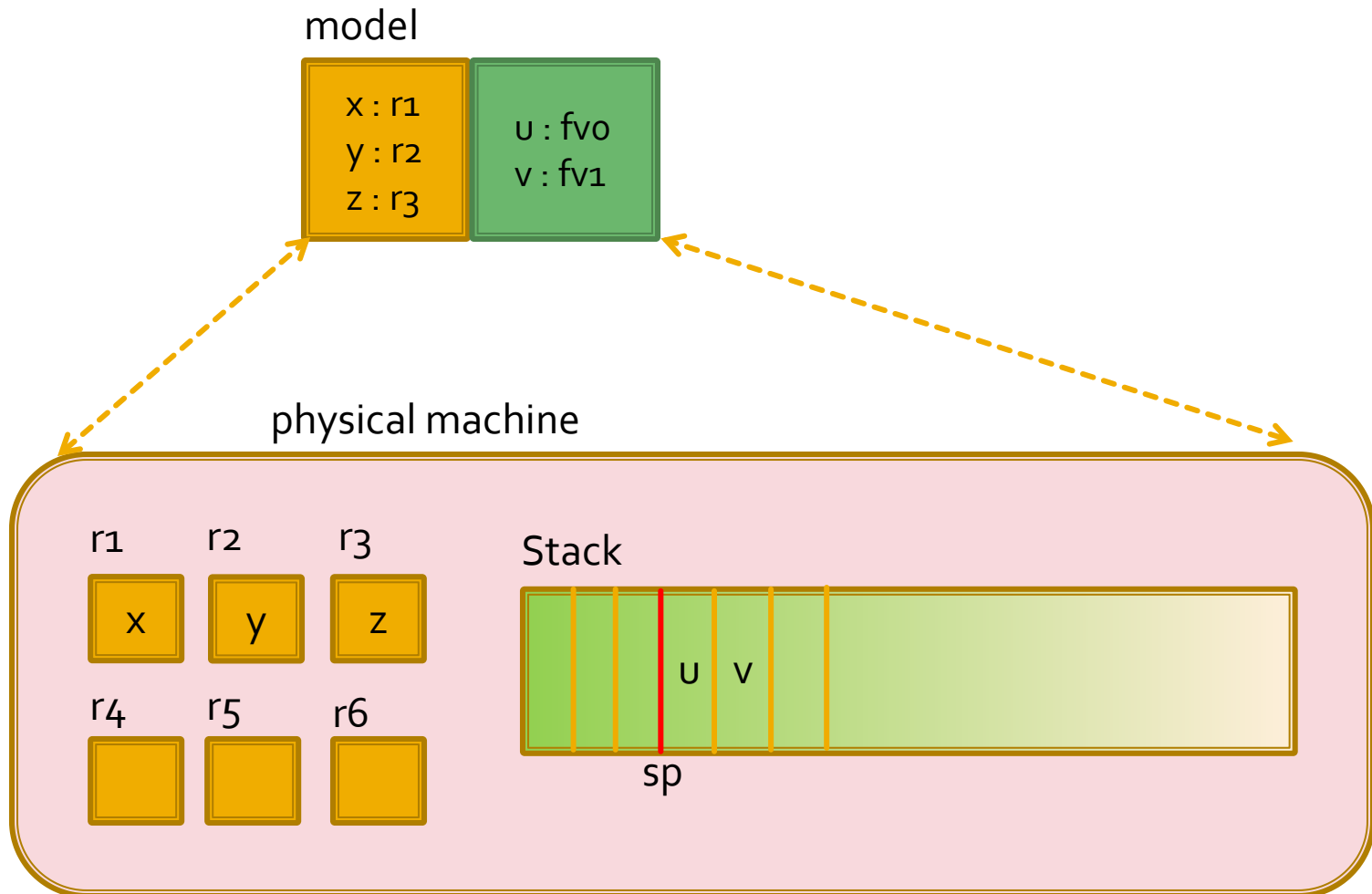
- What is a model?
 - It is a table which tells you where each name lives.
-
- Does it keep track of register and stack separately?
 - Yes, for maximum accuracy.
-

- Can you show me an example model?

- There you are!

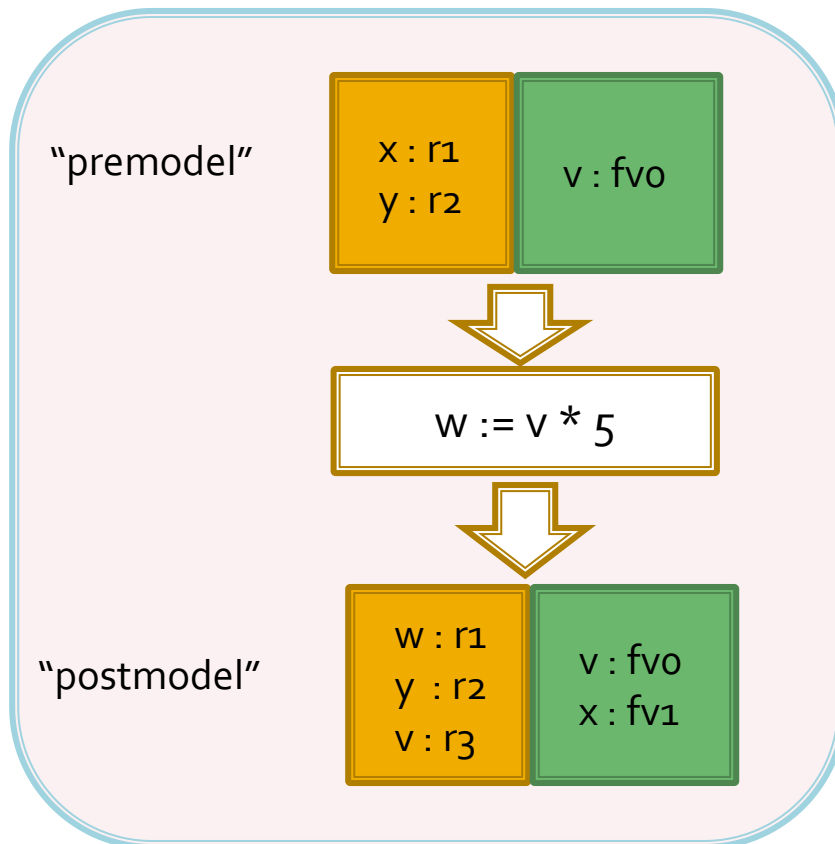
x : r1	u : fv0
y : r2	v : fv1
z : r3	

Model and Reality



Model Transformer Semantics

- How do models relate to each other?



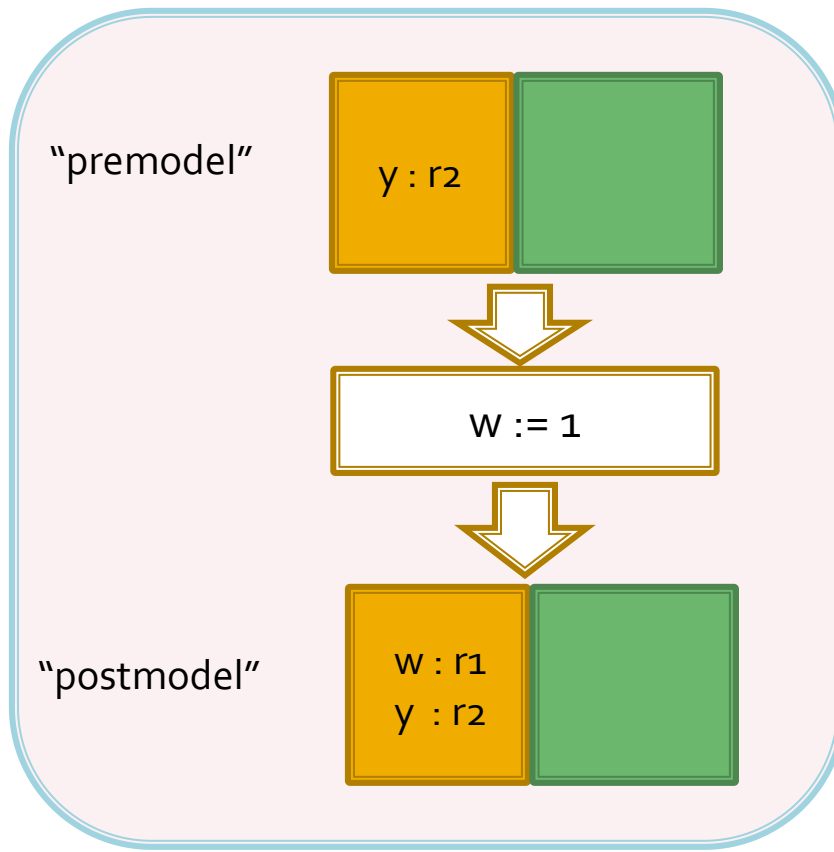
- Models relate to each other according to some inference rules, similar to that of Hoare Logic.

as Hoare-triple-style notation:

$$\{(x : r_1, y : r_2)(v : fv_0)\}$$
$$w := v * 5$$
$$\{(w : r_1, y : r_2, v : r_3)(v : fv_0, x : fv_1)\}$$

Births of Names

- When are names created?
- When they are assigned a value.



Deaths of Names

- Will names ever die?

- Yes. When we no longer refer to them.

-
- Where does x die?

```
x := 1
y := x + 1
z := y * 2
END
```

- After x is added with 1, but before assigning to y . I mean, here:


 $y := x + 1$

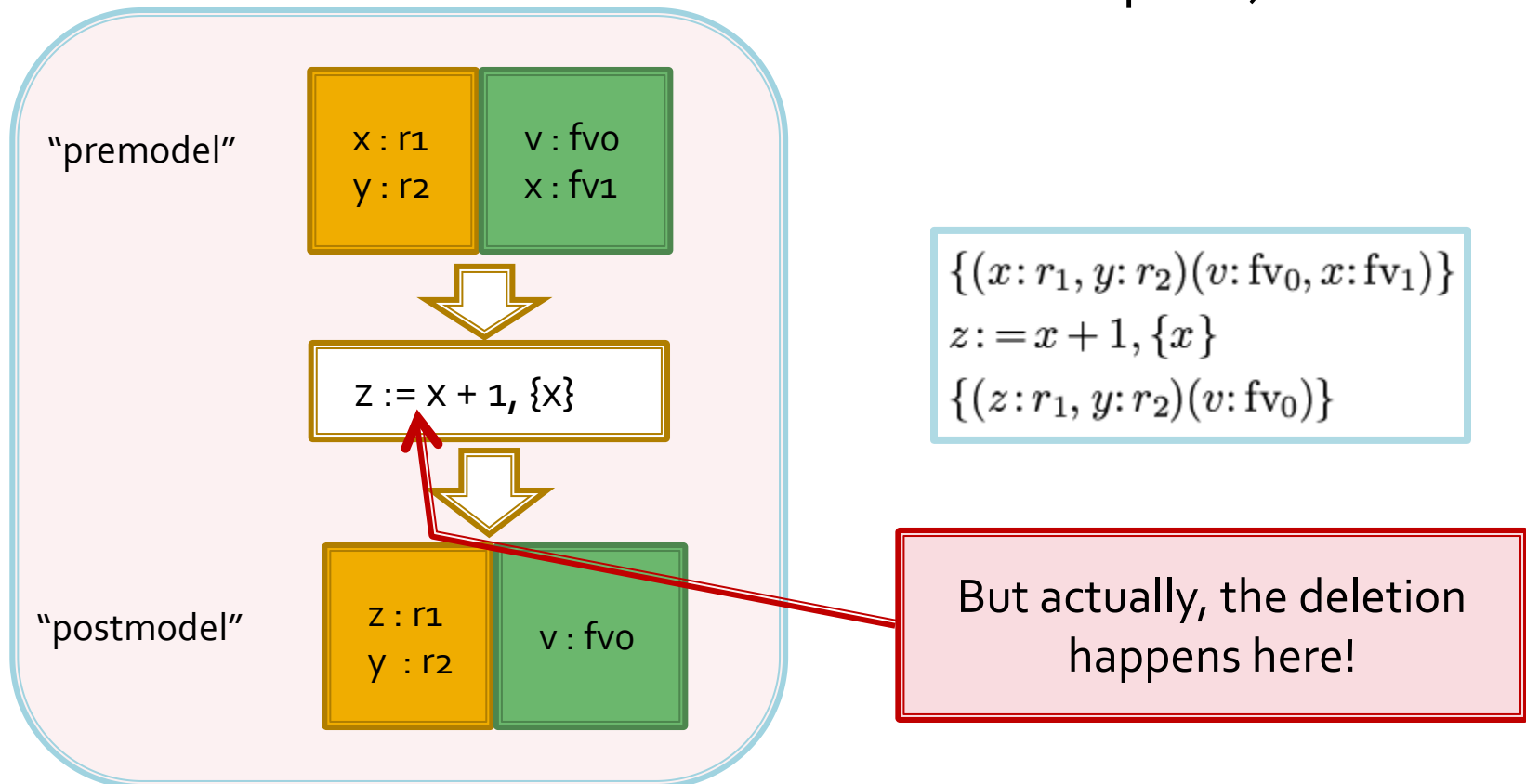
-
- How do we signify where a variable dies?

- We mark their endings, like:

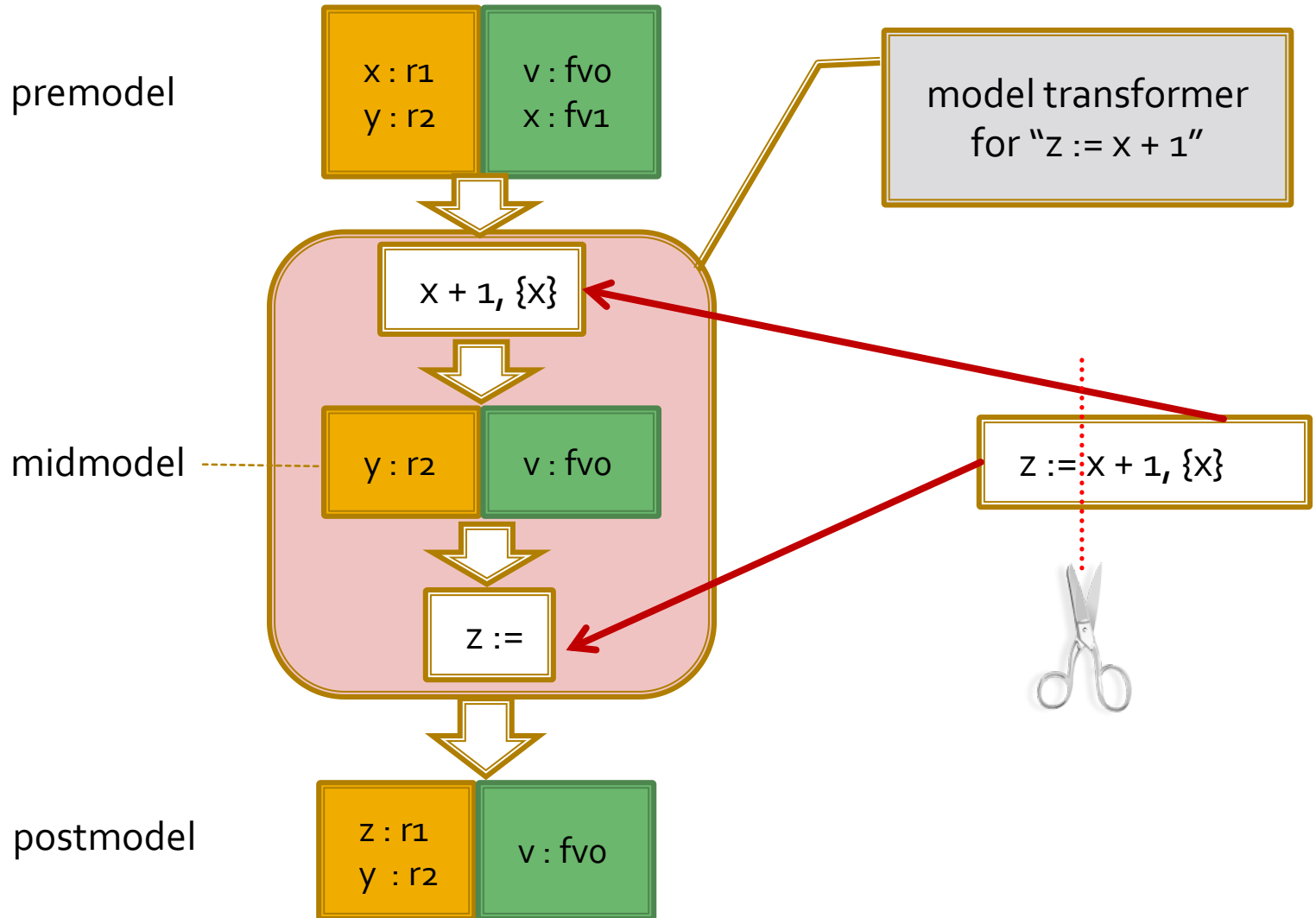
$y := x + 1, \{x\}$

Deletion in Model

- What happens to a model when a name dies?
- It is deleted from the model (for both register and stack parts).



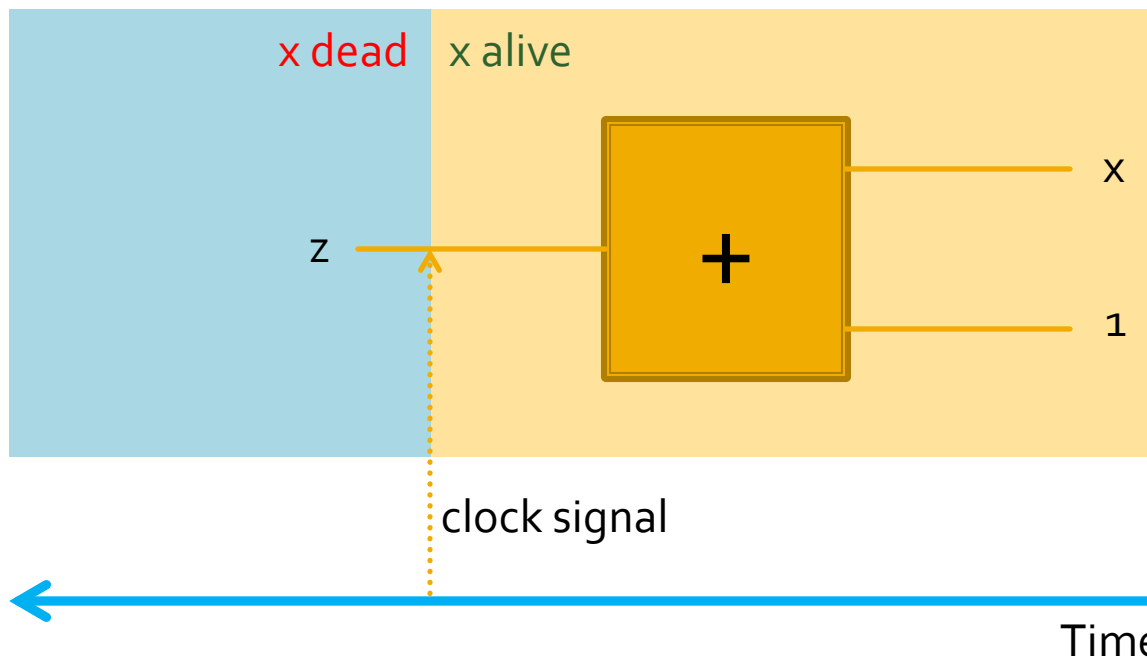
Midmodel



Why midmodel?

- Why is there a midmodel?

- Because the instruction must take at least two cycles to finish. x is already dead in the second cycle.



This is why z can reuse the register of x .

```
z := x + 1, {x}
```



The Bigger Picture

Input Language

- Procedure definitions
 - Sequencing
 - Arithmetic
 - Assignments
 - Memory references
 - Branching
 - Calls (with trivial arguments)
- UIL (unified intermediate language) after removing complex operands (remove-complex-opera*)

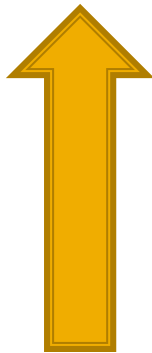
```

$$\begin{aligned} P &\rightarrow f: \lambda x y z. S * \\ S &\rightarrow s_1; S * \\ &| x \leftarrow y \\ &| x \leftarrow y + z \\ &| x \leftarrow [y + z] \\ &| [x + y] \leftarrow z \\ &| \text{if } t \text{ then } S * \text{ else } S * \\ &| f(x, y, z) \end{aligned}$$

```

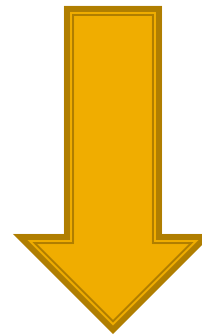
Two Pass Allocation

```
x := 1  
y := x + 1  
z := y * 2  
END
```



```
x := 1, {}  
y := x + 1, {x}  
z := y * 2, {y}  
END, {z}
```

Backward scan
for marking
deaths of names

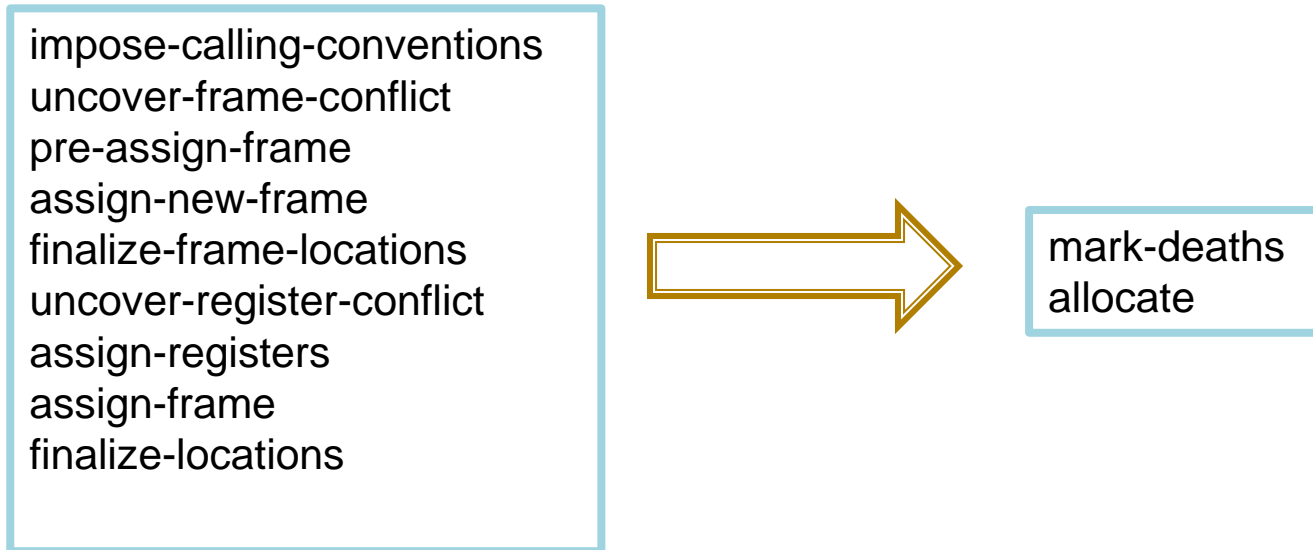


```
r1 := 1  
r1 := r1 + 1  
r1 := r1 * 2  
END
```

Forward model
transformation
and rewriting

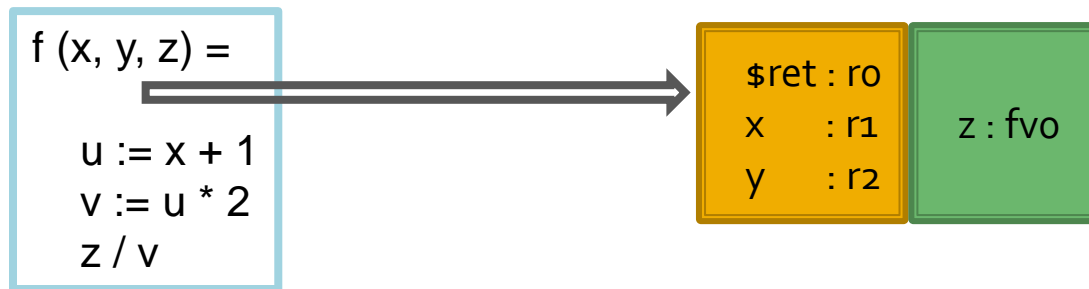
Reduction in compiler passes

- Nine passes of nanopass compiler reduced to two.



Starting Model

- What is the model at the beginning of a procedure?
- It is pre-determined by calling conventions.



Suppose r1, r2 are parameter registers, r0 is return address register. z has to be put into stack.

- What is `$ret` in the initial model?

- It is the return address set by the caller.

- What is the essence of it?

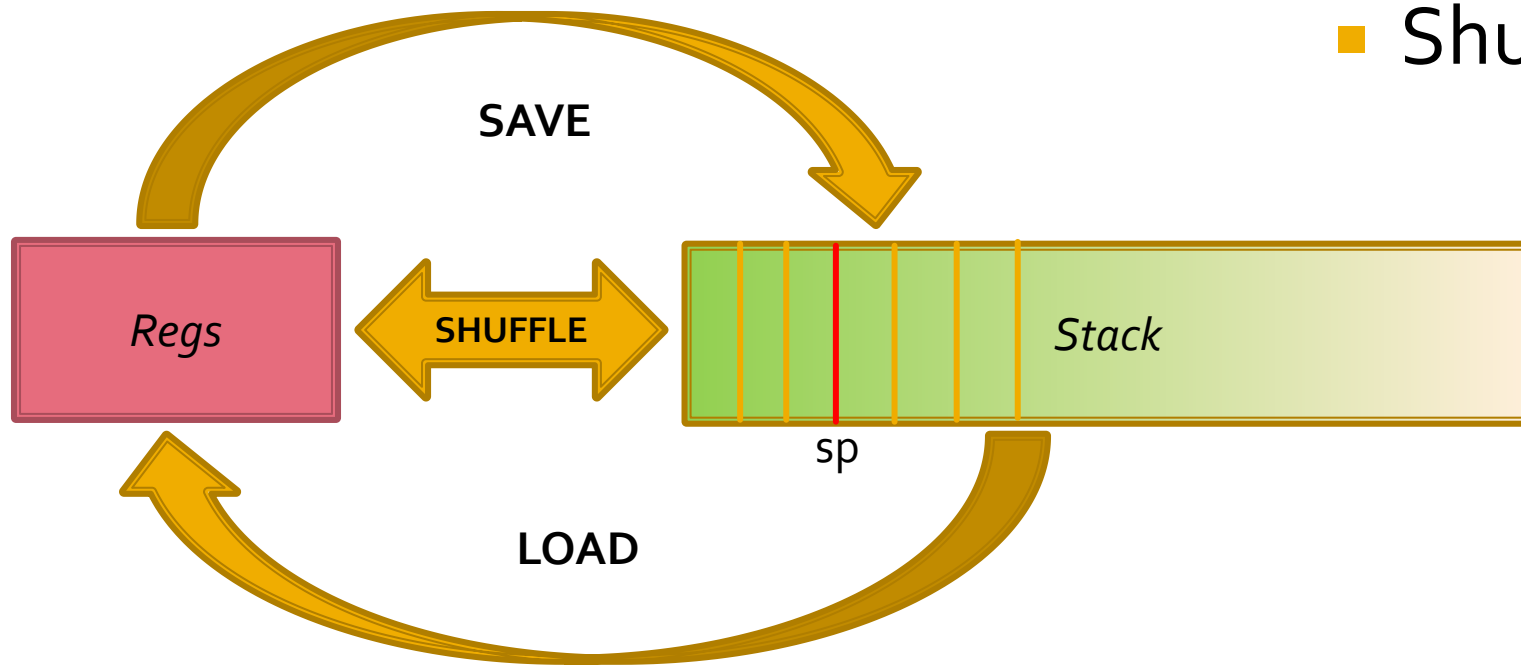
- It is the continuation “k” in a CPSed functional program:

```
f x y z k =  
  k (z / ((x + 1) * 2))
```

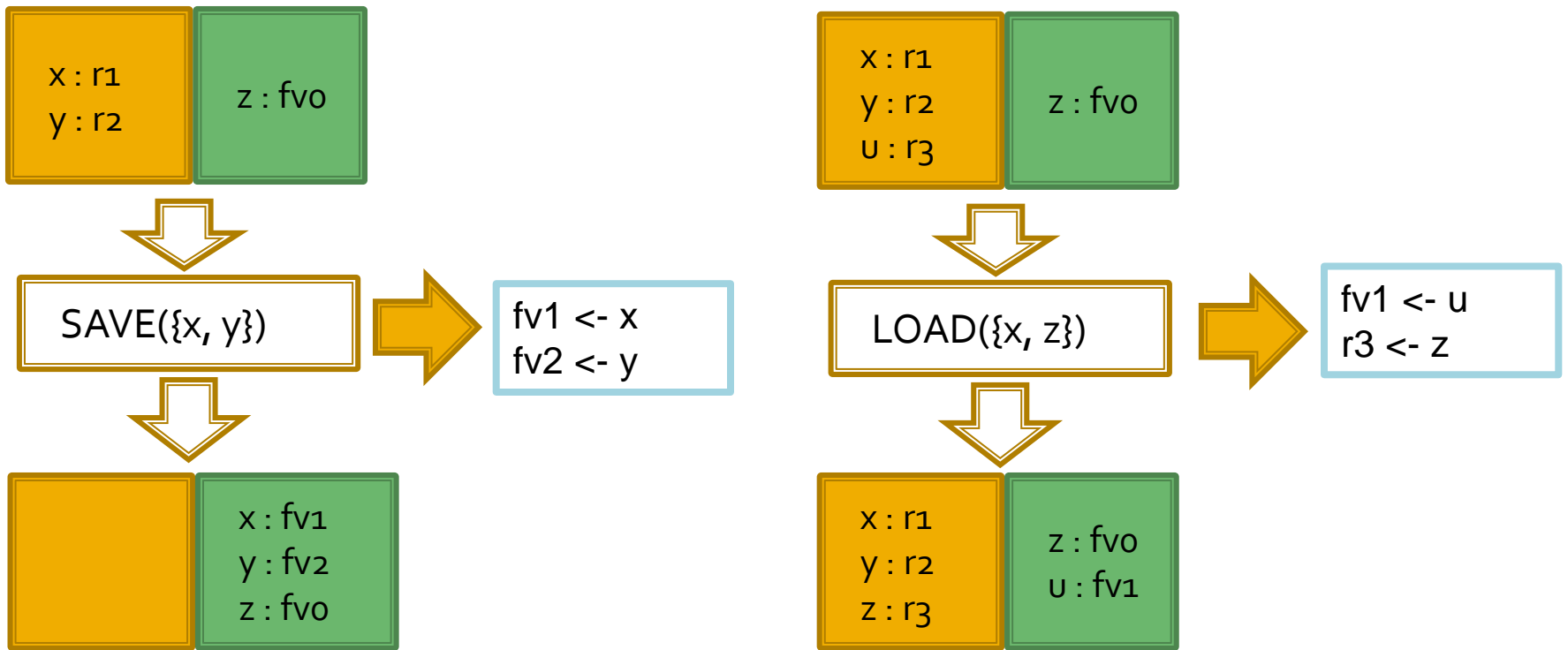
Primitive Model Transformers

- Transform the input model into new ones
- May emit instructions for loads and stores

- Save
- Load
- Shuffle



Examples



- What is the essence of the inserted instructions from SAVE and LOAD?

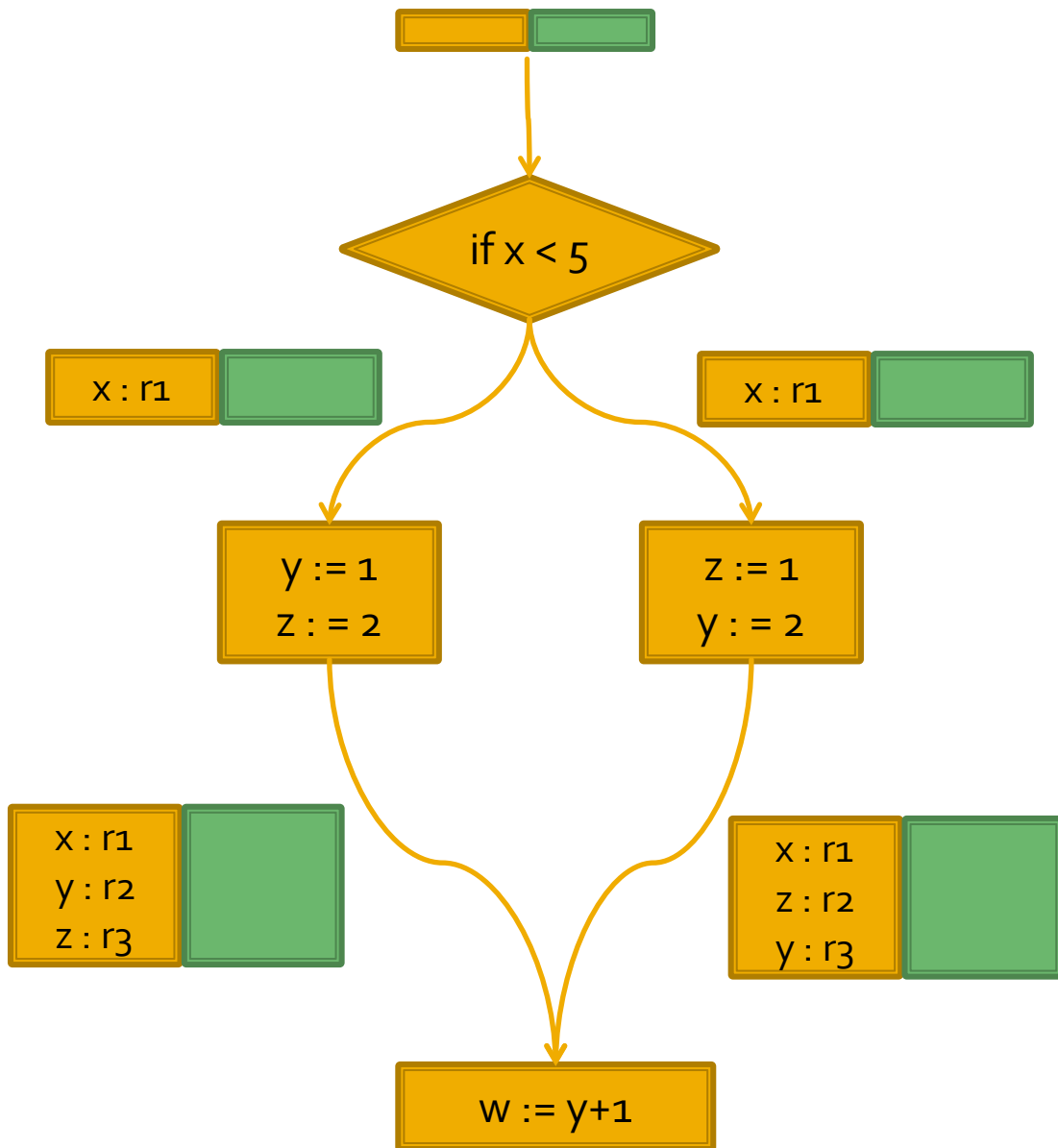
- They are actually doing “**live range splitting**” on-the-fly.

- What is the difference between splitting and spilling?

- Spilling puts the **whole** life span of a name in stack. Splitting only puts **part** of its life span in stack.

Branching

- What if different registers are assigned to the same variables in different branches
- We need to shuffle them for consistency.



Need shuffle:
r2 <-> r3

Shuffling

- Shuffling may happen at
 - Procedure calls
 - Join points
- Shuffling is like permutation

$r_1 \rightarrow r_2$

$r_2 \rightarrow r_3$

$r_3 \rightarrow r_1$

$r_4 \rightarrow r_5$

$r_5 \rightarrow r_4$

Handling Shuffling

- Break permutation into cycles (or paths):

$$(r_1 r_2 r_3) \circ (r_4 r_5)$$

- Generate code to move variables in cycles:

$$r_6 \leftarrow r_1$$

$$r_1 \leftarrow r_3$$

$$r_3 \leftarrow r_2$$

$$r_2 \leftarrow r_6$$

$$r_6 \leftarrow r_4$$

$$r_4 \leftarrow r_5$$

$$r_5 \leftarrow r_6$$

Related Work

Mostly graph coloring (extensions) and linear scan (extensions).

Assumptions of Graph Coloring

- A variable either lives in a register *throughout its life time*, or lives in a stack location *throughout its life time*
- A variable lives in the *same* register or stack location throughout its life time
- But actually,
 - A variable may live in a register, move to a stack location, and then move back to a (possibly different) register.
 - A variable may live in both a register AND a stack location.
- Graph Coloring doesn't capture this kind of semantics and may generate inefficient code.

Goal of Graph Coloring

- Goal: minimizing the number of allocated registers. (Doesn't aim at reducing memory traffic!)
- Produces good code only if no spilling happens. Otherwise no guarantee about memory traffic
- Spilled variables may introduce unpredictably many memory references

Is it still Graph Coloring?

- Still need backtracking for “optimal” allocation because of join points
- Maybe reducible to SAT (thus a graph), but a much more complex graph (may contain continuations of the allocator)
- Needs further investigation

Compare with Linear Scan

- It never spills, but linear scan spills.
- Does “**online** live range splitting”
 - existed partly in extended linear scan
 - usually done in a separate pass and thus *offline* (as in LLVM option “*pre-alloc-split*”)

Implications to Verified Compilation

- The model transformer semantics may be used to simplify formal verification of register allocation
- Formerly this is done by dynamic “validation” and not static verification (as in CompCert)

TODOs

- Handle instructions with memory operands and other irregularities
- Limited form of backtracking
- Callee-save registers and link-time optimization
- An industrial-strength compiler backend eventually

Acknowledgements

Thanks to R. Kent Dybvig, Andy Keep, Oleg Kiselyov for helpful comments and discussions.

Questions?

Thank you!

